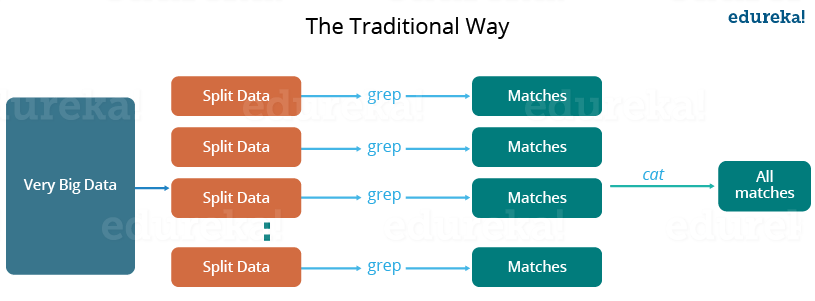
**MapReduce Tutorial: Introduction**

In this MapReduce Tutorial blog, I am going to introduce you to MapReduce, which is one of the core building blocks of processing in Hadoop framework. Before moving ahead, I would suggest you to get familiar with HDFS concepts which I have covered in my previous [***HDFS tutorial***](https://www.edureka.co/blog/apache-hadoop-hdfs-architecture/) blog. This will help you to understand the MapReduce concepts quickly and easily.

Google released a paper on MapReduce technology in December, 2004. This became the genesis of the Hadoop Processing Model. So, MapReduce is a programming model that allows us to perform parallel and distributed processing on huge data sets. The topics that I have covered in this MapReduce tutorial blog are as follows:

* [**Traditional Way for parallel and distributed processing**](https://www.edureka.co/blog/mapreduce-tutorial/#traditional_way)
* [**What is MapReduce?**](https://www.edureka.co/blog/mapreduce-tutorial/#what_is_mapreduce)
* [**MapReduce Example**](https://www.edureka.co/blog/mapreduce-tutorial/#mapreduce_word_count_example)
* [**MapReduce Advantages**](https://www.edureka.co/blog/mapreduce-tutorial/#mapreduce_advantages)
* [**MapReduce Program**](https://www.edureka.co/blog/mapreduce-tutorial/#mapreduce_example_program)
* [**MapReduce Program Explained**](https://www.edureka.co/blog/mapreduce-tutorial/#explanation_of_mapreduce_program)

**MapReduce Tutorial: Traditional Way**



Let us understand, when the MapReduce framework was not there, how parallel and distributed processing used to happen in a traditional way. So, let us take an example where I have a weather log containing the daily average temperature of the years from 2000 to 2015. Here, I want to calculate the day having the highest temperature in each year.

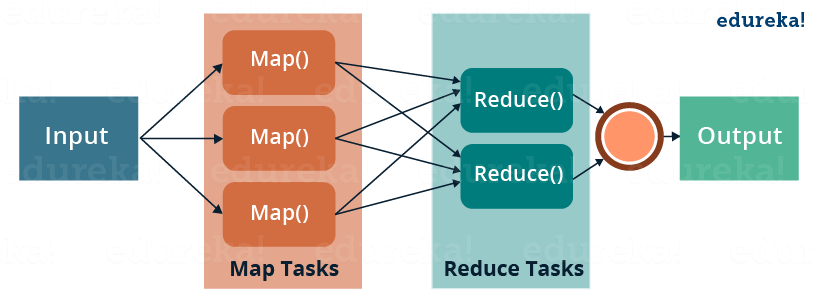
So, just like in the traditional way, I will split the data into smaller parts or blocks and store them in different machines. Then, I will find the highest temperature in each part stored in the corresponding machine. At last, I will combine the results received from each of the machines to have the final output. Let us look at the challenges associated with this traditional approach:

1. **Critical path problem:** It is the amount of time taken to finish the job without delaying the next milestone or actual completion date. So, if, any of the machines delays the job, the whole work gets delayed.
2. **Reliability problem:** What if, any of the machines which is working with a part of data fails? The management of this failover becomes a challenge.
3. **Equal split issue:** How will I divide the data into smaller chunks so that each machine gets even part of data to work with. In other words, how to equally divide the data such that no individual machine is overloaded or under utilized.
4. **Single split may fail:** If any of the machine fails to provide the output, I will not be able to calculate the result. So, there should be a mechanism to ensure this fault tolerance capability of the system.
5. **Aggregation of result:** There should be a mechanism to aggregate the result generated by each of the machines to produce the final output.

These are the issues which I will have to take care individually while performing parallel processing of huge data sets when using traditional approaches.

To overcome these issues, we have the MapReduce framework which allows us to perform such parallel computations without bothering about the issues like reliability, fault tolerance etc. Therefore, MapReduce gives you the flexibility to write code logic without caring about the design issues of the system.

**MapReduce Tutorial: What is MapReduce?**



MapReduce is a programming framework that allows us to perform distributed and parallel processing on large data sets in a distributed environment.

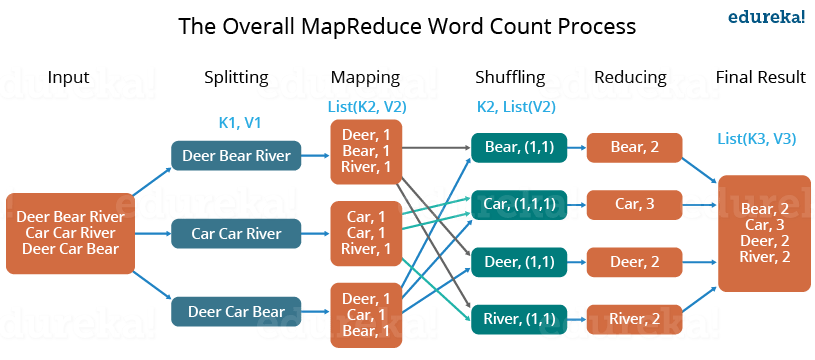
* MapReduce consists of two distinct tasks – Map and Reduce.
* As the name MapReduce suggests, reducer phase takes place after mapper phase has been completed.
* So, the first is the map job, where a block of data is read and processed to produce key-value pairs as intermediate outputs.
* The output of a Mapper or map job (key-value pairs) is input to the Reducer.
* The reducer receives the key-value pair from multiple map jobs.
* Then, the reducer aggregates those intermediate data tuples (intermediate key-value pair) into a smaller set of tuples or key-value pairs which is the final output.

**MapReduce Tutorial: A Word Count Example of MapReduce**

Let us understand, how a MapReduce works by taking an example where I have a text file called example.txt whose contents are as follows:

**Dear, Bear, River, Car, Car, River, Deer, Car and Bear**

Now, suppose, we have to perform a word count on the sample.txt using MapReduce. So, we will be finding the unique words and the number of occurrences of those unique words.



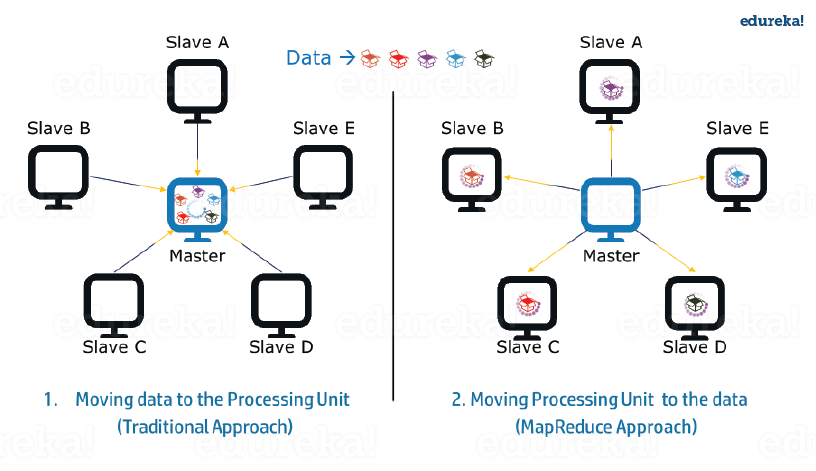
* First, we divide the input in three splits as shown in the figure. This will distribute the work among all the map nodes.
* Then, we tokenize the words in each of the mapper and give a hardcoded value (1) to each of the tokens or words. The rationale behind giving a hardcoded value equal to 1 is that every word, in itself, will occur once.
* Now, a list of key-value pair will be created where the key is nothing but the individual words and value is one. So, for the first line (Dear Bear River) we have 3 key-value pairs – Dear, 1; Bear, 1; River, 1. The mapping process remains the same on all the nodes.
* After mapper phase, a partition process takes place where sorting and shuffling happens so that all the tuples with the same key are sent to the corresponding reducer.
* So, after the sorting and shuffling phase, each reducer will have a unique key and a list of values corresponding to that very key. For example, Bear, [1,1]; Car, [1,1,1].., etc.
* Now, each Reducer counts the values which are present in that list of values. As shown in the figure, reducer gets a list of values which is [1,1] for the key Bear. Then, it counts the number of ones in the very list and gives the final output as – Bear, 2.
* Finally, all the output key/value pairs are then collected and written in the output file.

**MapReduce Tutorial: Advantages of MapReduce**

The two biggest advantages of MapReduce are:

**1. Parallel Processing:**

In MapReduce, we are dividing the job among multiple nodes and each node works with a part of the job simultaneously. So, MapReduce is based on Divide and Conquer paradigm which helps us to process the data using different machines. As the data is processed by multiple machine instead of a single machine in parallel, the time taken to process the data gets reduced by a tremendous amount as shown in the figure below (2).

***Fig.:*** Traditional Way Vs. MapReduce Way – MapReduce Tutorial

**2. Data Locality:**

Instead of moving data to the processing unit, we are moving processing unit to the data in the MapReduce Framework.  In the traditional system, we used to bring data to the processing unit and process it. But, as the data grew and became very huge, bringing this huge amount of data to the processing unit posed following issues:

* Moving huge data to processing is costly and deteriorates the network performance.
* Processing takes time as the data is processed by a single unit which becomes the bottleneck.
* Master node can get over-burdened and may fail.

Now, MapReduce allows us to overcome above issues by bringing the processing unit to the data. So, as you can see in the above image that the data is distributed among multiple nodes where each node processes the part of the data residing on it. This allows us to have the following advantages:

Next

* It is very cost effective to move processing unit to the data.
* The processing time is reduced as all the nodes are working with their part of the data in parallel.
* Every node gets a part of the data to process and therefore, there is no chance of a node getting overburdened.

**MapReduce Tutorial: MapReduce Example Program**

Before jumping into the details, let us have a glance at a MapReduce example program to have a basic idea about how things work in a MapReduce environment practically. I have taken the same word count example where I have to find out the number of occurrences of each word. And Don’t worry guys, if you don’t understand the code when you look at it for the first time, just bear with me while I walk you through each part of the MapReduce code.

**Source code:**

package co.edureka.mapreduce;

import java.io.IOException;

import java.util.StringTokenizer;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.fs.Path;

public class WordCount

{

public static class Map extends Mapper<LongWritable,Text,Text,IntWritable> {

public void map(LongWritable key, Text value,Context context) throws IOException,

InterruptedException{

String line = value.toString();

StringTokenizer tokenizer = new StringTokenizer(line);

while (tokenizer.hasMoreTokens()) {

value.set(tokenizer.nextToken());

context.write(value, new IntWritable(1));

}

}

}

public static class Reduce extends Reducer<Text,IntWritable,Text,IntWritable> {

public void reduce(Text key, Iterable<IntWritable> values,Context context) throws

IOException,InterruptedException {

int sum=0;

for(IntWritable x: values)

{

sum+=x.get();

}

context.write(key, new IntWritable(sum));

}

}

public static void main(String[] args) throws Exception {

Configuration conf= new Configuration();

Job job = new Job(conf,"My Word Count Program");

job.setJarByClass(WordCount.class);

job.setMapperClass(Map.class);

job.setReducerClass(Reduce.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

job.setInputFormatClass(TextInputFormat.class);

job.setOutputFormatClass(TextOutputFormat.class);

Path outputPath = new Path(args[1]);

//Configuring the input/output path from the filesystem into the job

FileInputFormat.addInputPath(job, new Path(args[0]));

FileOutputFormat.setOutputPath(job, new Path(args[1]));

//deleting the output path automatically from hdfs so that we don't have to

delete it explicitly

outputPath.getFileSystem(conf).delete(outputPath);

//exiting the job only if the flag value becomes false

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

}

**MapReduce Tutorial: Explanation of MapReduce Program**

The entire MapReduce program can be fundamentally divided into three parts:

* Mapper Phase Code
* Reducer Phase Code
* Driver Code

We will understand the code for each of these three parts sequentially.

**Mapper code:**

public static class Map extends Mapper<LongWritable,Text,Text,IntWritable> {

public void map(LongWritable key, Text value, Context context) throws IOException,InterruptedException {

String line = value.toString();

StringTokenizer tokenizer = new StringTokenizer(line);

while (tokenizer.hasMoreTokens()) {

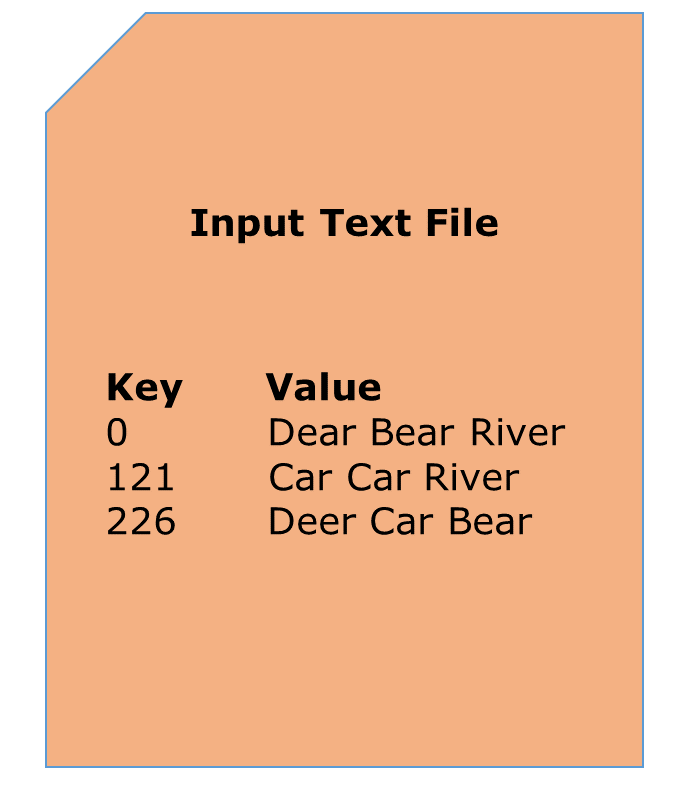
value.set(tokenizer.nextToken());

context.write(value, new IntWritable(1));

}

}

}

We have created a class Map that extends the class Mapper which is already defined in the MapReduce Framework.

* We define the data types of input and output key/value pair after the class declaration using angle brackets.
* Both the input and output of the Mapper is a key/value pair.
* Input:
  + The *key* is nothing but the offset of each line in the text file: *LongWritable*
  + The *value* is each individual line (as shown in the figure at the right): *Text*
* Output:
  + The *key* is the tokenized words: *Text*
  + We have the hardcoded *value* in our case which is 1: *IntWritable*
  + Example – Dear 1, Bear 1, etc.
* We have written a java code where we have tokenized each word and assigned them a hardcoded value equal to *1*.

**Reducer Code:**

public static class Reduce extends Reducer<Text,IntWritable,Text,IntWritable> {

public void reduce(Text key, Iterable<IntWritable> values,Context context)

throws IOException,InterruptedException {

int sum=0;

for(IntWritable x: values)

{

sum+=x.get();

}

context.write(key, new IntWritable(sum));

}

}

* We have created a class Reduce which extends class Reducer like that of Mapper.
* We define the data types of input and output key/value pair after the class declaration using angle brackets as done for Mapper.
* Both the input and the output of the Reducer is a key-value pair.
* Input:
  + The *key* nothing but those unique words which have been generated after the sorting and shuffling phase: *Text*
  + The *value* is a list of integers corresponding to each key: *IntWritable*
  + Example – Bear, [1, 1], etc.
* Output:
  + The *key* is all the unique words present in the input text file: *Text*
  + The *value* is the number of occurrences of each of the unique words: *IntWritable*
  + Example – Bear, 2; Car, 3, etc.
* We have aggregated the values present in each of the list corresponding to each key and produced the final answer.
* In general, a single reducer is created for each of the unique words, but, you can specify the number of reducer in mapred-site.xml.

**Driver Code:**

Configuration conf= new Configuration();

Job job = new Job(conf,"My Word Count Program");

job.setJarByClass(WordCount.class);

job.setMapperClass(Map.class);

job.setReducerClass(Reduce.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

job.setInputFormatClass(TextInputFormat.class);

job.setOutputFormatClass(TextOutputFormat.class);

Path outputPath = new Path(args[1]);

//Configuring the input/output path from the filesystem into the job

FileInputFormat.addInputPath(job, new Path(args[0]));

FileOutputFormat.setOutputPath(job, new Path(args[1]));

* In the driver class, we set the configuration of our MapReduce job to run in Hadoop.
* We specify the name of the job , the data type of input/output of the mapper and reducer.
* We also specify the names of the mapper and reducer classes.
* The path of the input and output folder is also specified.
* The method setInputFormatClass () is used for specifying that how a Mapper will read the input data or what will be the unit of work. Here, we have chosen TextInputFormat so that single line is read by the mapper at a time from the input text file.
* The main () method is the entry point for the driver. In this method, we instantiate a new Configuration object for the job.

**Run the MapReduce code:**

The command for running a MapReduce code is:

*hadoop jar hadoop-mapreduce-example.jar WordCount /sample/input /sample/output*

**How MapReduce Works? Complete Process**

The whole process goes through four phases of execution namely, splitting, mapping, shuffling, and reducing.

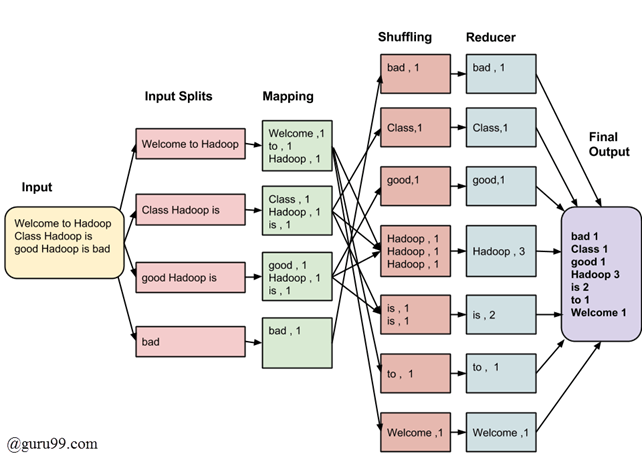
Let's understand this with an example –

Consider you have following input data for your Map Reduce Program

Welcome to Hadoop Class

Hadoop is good

Hadoop is bad

[](https://www.guru99.com/images/Big_Data/061114_0930_Introductio1.png)

MapReduce Architecture

The final output of the MapReduce task is

bad 1

Class 1

good 1

Hadoop 3

is 2

to 1

Welcome 1

The data goes through the following phases

**Input Splits:**

An input to a MapReduce job is divided into fixed-size pieces called **input splits**Input split is a chunk of the input that is consumed by a single map

**Mapping**

This is the very first phase in the execution of map-reduce program. In this phase data in each split is passed to a mapping function to produce output values. In our example, a job of mapping phase is to count a number of occurrences of each word from input splits (more details about input-split is given below) and prepare a list in the form of <word, frequency>

**Shuffling**

This phase consumes the output of Mapping phase. Its task is to consolidate the relevant records from Mapping phase output. In our example, the same words are clubed together along with their respective frequency.

**Reducing**

In this phase, output values from the Shuffling phase are aggregated. This phase combines values from Shuffling phase and returns a single output value. In short, this phase summarizes the complete dataset.

In our example, this phase aggregates the values from Shuffling phase i.e., calculates total occurrences of each word.

**MapReduce Architecture explained in detail**

* One map task is created for each split which then executes map function for each record in the split.
* It is always beneficial to have multiple splits because the time taken to process a split is small as compared to the time taken for processing of the whole input. When the splits are smaller, the processing is better to load balanced since we are processing the splits in parallel.
* However, it is also not desirable to have splits too small in size. When splits are too small, the overload of managing the splits and map task creation begins to dominate the total job execution time.
* For most jobs, it is better to make a split size equal to the size of an HDFS block (which is 64 MB, by default).
* Execution of map tasks results into writing output to a local disk on the respective node and not to HDFS.
* Reason for choosing local disk over HDFS is, to avoid replication which takes place in case of HDFS store operation.
* Map output is intermediate output which is processed by reduce tasks to produce the final output.
* Once the job is complete, the map output can be thrown away. So, storing it in HDFS with replication becomes overkill.
* In the event of node failure, before the map output is consumed by the reduce task, Hadoop reruns the map task on another node and re-creates the map output.
* Reduce task doesn't work on the concept of data locality. An output of every map task is fed to the reduce task. Map output is transferred to the machine where reduce task is running.
* On this machine, the output is merged and then passed to the user-defined reduce function.
* Unlike the map output, reduce output is stored in HDFS (the first replica is stored on the local node and other replicas are stored on off-rack nodes). So, writing the reduce output

**How MapReduce Organizes Work?**

Hadoop divides the job into tasks. There are two types of tasks:

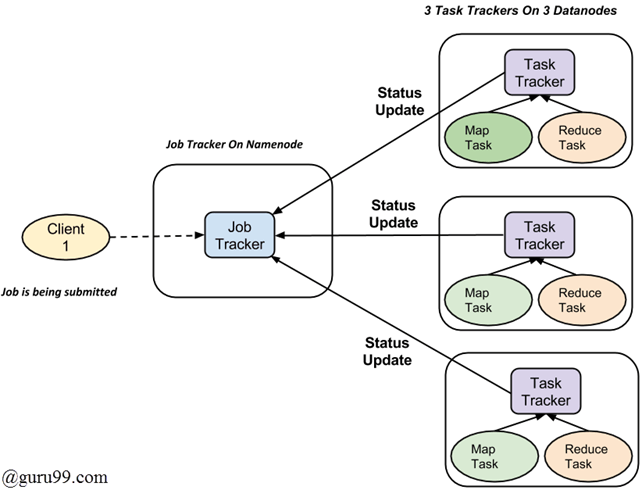
1. **Map tasks** (Splits & Mapping)
2. **Reduce tasks** (Shuffling, Reducing)

as mentioned above.

The complete execution process (execution of Map and Reduce tasks, both) is controlled by two types of entities called a

1. **Jobtracker**: Acts like a **master** (responsible for complete execution of submitted job)
2. **Multiple Task Trackers**: Acts like **slaves,** each of them performing the job

For every job submitted for execution in the system, there is one **Jobtracker** that resides on **Namenode** and there are **multiple tasktrackers** which reside on **Datanode**.

[](https://www.guru99.com/images/Big_Data/061114_0930_Introductio2.png)

* A job is divided into multiple tasks which are then run onto multiple data nodes in a cluster.
* It is the responsibility of job tracker to coordinate the activity by scheduling tasks to run on different data nodes.
* Execution of individual task is then to look after by task tracker, which resides on every data node executing part of the job.
* Task tracker's responsibility is to send the progress report to the job tracker.
* In addition, task tracker periodically sends **'heartbeat'** signal to the Jobtracker so as to notify him of the current state of the system.
* Thus job tracker keeps track of the overall progress of each job. In the event of task failure, the job tracker can reschedule it on a different task tracker.

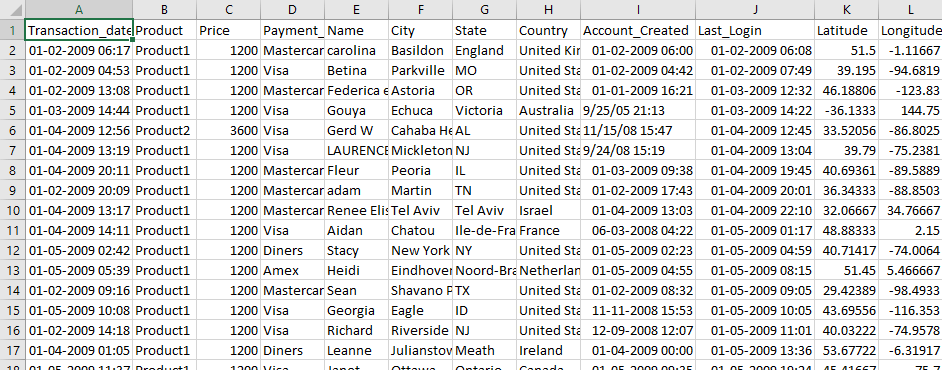
# Hadoop & Mapreduce Examples: Create your First Program

In this tutorial, you will learn to use Hadoop and MapReduce with Example. The input data used is [SalesJan2009.csv](https://drive.google.com/uc?export=download&id=1tP8AJGSgDXwI12r2Ap07GyamMj1o0iDD). It contains Sales related information like Product name, price, payment mode, city, country of client etc. The goal is to ***Find out Number of Products Sold in Each Country.***

In this tutorial, you will learn-

* [First Hadoop MapReduce Program](https://www.guru99.com/create-your-first-hadoop-program.html#1)
* [Explanation of SalesMapper Class](https://www.guru99.com/create-your-first-hadoop-program.html#2)
* [Explanation of SalesCountryReducer Class](https://www.guru99.com/create-your-first-hadoop-program.html#3)
* [Explanation of SalesCountryDriver Class](https://www.guru99.com/create-your-first-hadoop-program.html#4)

### First Hadoop MapReduce Program

[](https://www.guru99.com/images/1/sales-jan-2009.png)

Data of SalesJan2009

Ensure you have Hadoop installed

**SalesMapper.java**

package SalesCountry;

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.\*;

public class SalesMapper extends MapReduceBase implements Mapper <LongWritable, Text, Text, IntWritable> {

private final static IntWritable one = new IntWritable(1);

public void map(LongWritable key, Text value, OutputCollector <Text, IntWritable> output, Reporter reporter) throws IOException {

String valueString = value.toString();

String[] SingleCountryData = valueString.split(",");

output.collect(new Text(SingleCountryData[7]), one);

}

}

**SalesCountryReducer.java**

package SalesCountry;

import java.io.IOException;

import java.util.\*;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.\*;

public class SalesCountryReducer extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {

public void reduce(Text t\_key, Iterator<IntWritable> values, OutputCollector<Text,IntWritable> output, Reporter reporter) throws IOException {

Text key = t\_key;

int frequencyForCountry = 0;

while (values.hasNext()) {

// replace type of value with the actual type of our value

IntWritable value = (IntWritable) values.next();

frequencyForCountry += value.get();

}

output.collect(key, new IntWritable(frequencyForCountry));

}

}

**SalesCountryDriver.java**

package SalesCountry;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.\*;

import org.apache.hadoop.mapred.\*;

public class SalesCountryDriver {

public static void main(String[] args) {

JobClient my\_client = new JobClient();

// Create a configuration object for the job

JobConf job\_conf = new JobConf(SalesCountryDriver.class);

// Set a name of the Job

job\_conf.setJobName("SalePerCountry");

// Specify data type of output key and value

job\_conf.setOutputKeyClass(Text.class);

job\_conf.setOutputValueClass(IntWritable.class);

// Specify names of Mapper and Reducer Class

job\_conf.setMapperClass(SalesCountry.SalesMapper.class);

job\_conf.setReducerClass(SalesCountry.SalesCountryReducer.class);

// Specify formats of the data type of Input and output

job\_conf.setInputFormat(TextInputFormat.class);

job\_conf.setOutputFormat(TextOutputFormat.class);

// Set input and output directories using command line arguments,

//arg[0] = name of input directory on HDFS, and arg[1] = name of output directory to be created to store the output file.

FileInputFormat.setInputPaths(job\_conf, new Path(args[0]));

FileOutputFormat.setOutputPath(job\_conf, new Path(args[1]));

my\_client.setConf(job\_conf);

try {

// Run the job

JobClient.runJob(job\_conf);

} catch (Exception e) {

e.printStackTrace();

}

}

}

## MapReduce Algorithm

MapReduce is a Distributed Data Processing Algorithm, introduced by Google in it’s MapReduce Tech Paper.

MapReduce Algorithm is mainly inspired by Functional Programming model. ( Please read this post “[Functional Programming Basics](https://www.journaldev.com/8693/functional-imperative-object-oriented-programming-comparison)” to get some understanding about Functional Programming , how it works and it’s major advantages).

MapReduce algorithm is mainly useful to process huge amount of data in parallel, reliable and efficient way in cluster environments.

I hope, everyone is familiar with **“Divide and Conquer”** algorithm. It uses Divide and Conquer technique to process large amount of data.

It divides input task into smaller and manageable sub-tasks (They should be executable independently) to execute them in-parallel.

### MapReduce Algorithm Steps

MapReduce Algorithm uses the following three main steps:

1. Map Function
2. Shuffle Function
3. Reduce Function

Here we are going to discuss each function role and responsibility in MapReduce algorithm. If you don’t understand it well in this section, don’t get panic. Please read next section, where we use one simple word counting example to explain them in-detail. Once you read next section again come back to this section re-read it again. I bet you will definitely understand these 3 steps or functions very well.

### Map Function

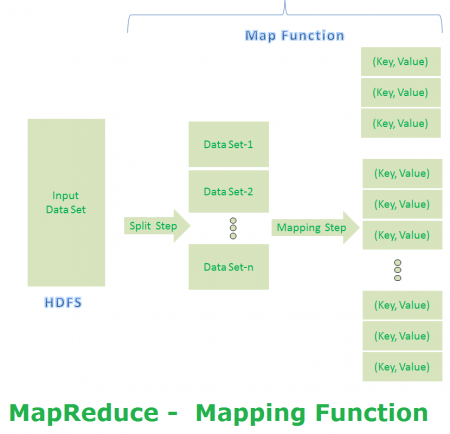
Map Function is the first step in MapReduce Algorithm. It takes input tasks (say DataSets. I have given only one DataSet in below diagram.) and divides them into smaller sub-tasks. Then perform required computation on each sub-task in parallel.

This step performs the following two sub-steps:

1. Splitting
2. Mapping

* Splitting step takes input DataSet from Source and divide into smaller Sub-DataSets.
* Mapping step takes those smaller Sub-DataSets and perform required action or computation on each Sub-DataSet.

The output of this Map Function is a set of key and value pairs as <Key, Value> as shown in the below diagram.

[](https://cdn.journaldev.com/wp-content/uploads/2015/09/mrv1-mapping.png)

**MapReduce First Step Output:**

[](https://cdn.journaldev.com/wp-content/uploads/2015/09/mapfunction-output.png)

### Shuffle Function

It is the second step in MapReduce Algorithm. Shuffle Function is also know as “Combine Function”.

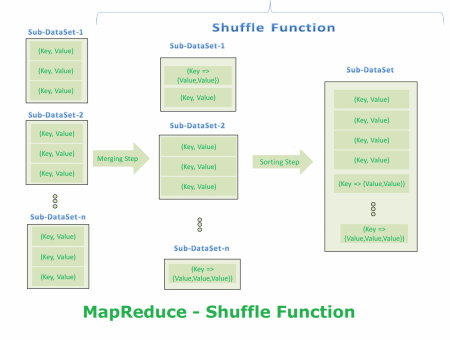
It performs the following two sub-steps:

1. Merging
2. Sorting

It takes a list of outputs coming from “Map Function” and perform these two sub-steps on each and every key-value pair.

* Merging step combines all key-value pairs which have same keys (that is grouping key-value pairs by comparing “Key”). This step returns <Key, List<Value>>.
* Sorting step takes input from Merging step and sort all key-value pairs by using Keys. This step also returns <Key, List<Value>> output but with sorted key-value pairs.

Finally, Shuffle Function returns a list of <Key, List<Value>> sorted pairs to next step.

[](https://cdn.journaldev.com/wp-content/uploads/2015/09/shuffle-function.png)

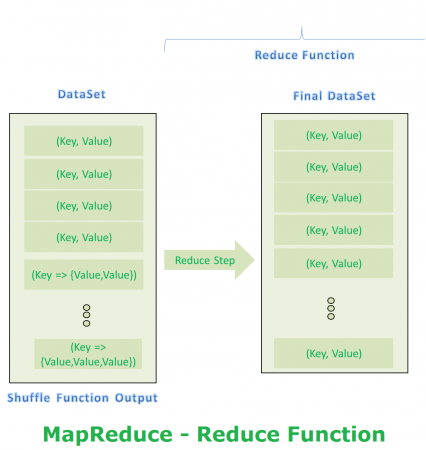
**MapReduce Second Step Output:**

[](https://cdn.journaldev.com/wp-content/uploads/2015/09/shuffle-function-output.png)

### Reduce Function

It is the final step in MapReduce Algorithm. It performs only one step : Reduce step.

It takes list of <Key, List<Value>> sorted pairs from Shuffle Function and perform reduce operation as shown below.

[](https://cdn.journaldev.com/wp-content/uploads/2015/09/reduce-function.png)

**MapReduce Final Step Output:**

[](https://cdn.journaldev.com/wp-content/uploads/2015/09/reduce-function-output.png)

Final step output looks like first step output. However final step <Key, Value> pairs are different than first step <Key, Value> pairs. Final step <Key, Value> pairs are computed and sorted pairs.

We can observe the difference between first step output and final step output with some simple example. We will discuss same steps with one simple example in next section.

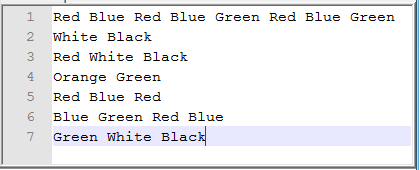
That’s it all three steps of MapReduce Algorithm.

### MapReduce Example – Word Count

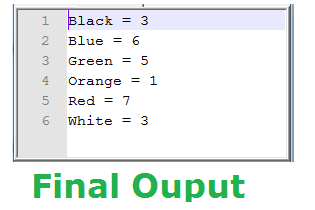
In this section, we are going to discuss about “How MapReduce Algorithm solves WordCount Problem” theoretically. We will implement a Hadoop MapReduce Program and test it in my coming post.

**Problem Statement:**  
Count the number of occurrences of each word available in a DataSet.

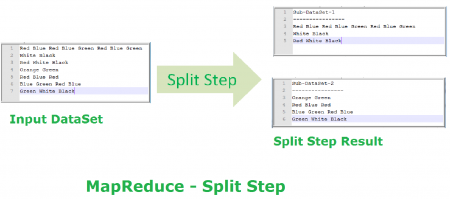
**Input DataSet**  
Please find our example Input DataSet file in below diagram. Just for simplicity, we are going to use simple small DataSet. However, Real-time applications use very huge amount of Data.

[](https://cdn.journaldev.com/wp-content/uploads/2015/09/wordcount-inputfile-content.png)

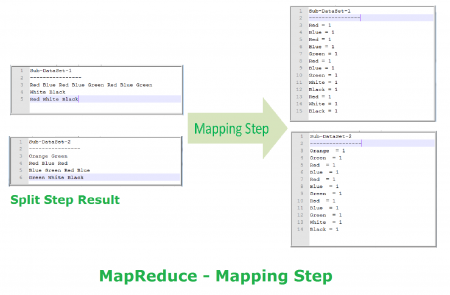
**Client Required Final Result**

[](https://cdn.journaldev.com/wp-content/uploads/2015/09/wordcount-final-output.png)

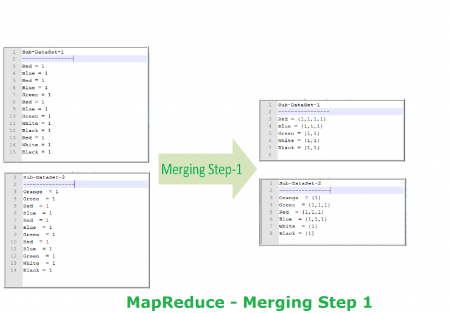
**MapReduce – Map Function (Split Step)**

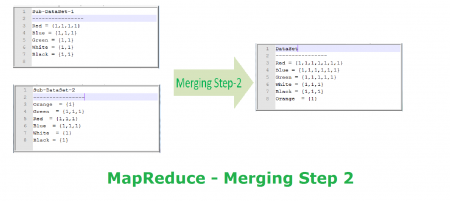
[](https://cdn.journaldev.com/wp-content/uploads/2015/09/wordcount-mapping-split-step.png)

**MapReduce – Map Function (Mapping Step)**

[](https://cdn.journaldev.com/wp-content/uploads/2015/09/wordcount-mapping-mapping-step.png)

**MapReduce – Shuffle Function (Merge Step)**

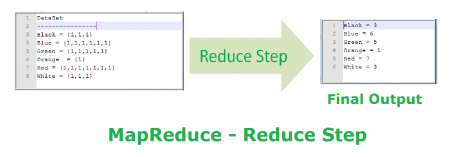
[](https://cdn.journaldev.com/wp-content/uploads/2015/09/wordcount-mapping-merge-step1.png)

[](https://cdn.journaldev.com/wp-content/uploads/2015/09/wordcount-mapping-merge-step2.png)

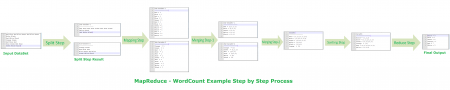
**MapReduce – Shuffle Function (Sorting Step)**

[](https://cdn.journaldev.com/wp-content/uploads/2015/09/wordcount-mapping-sorting-step.png)

**MapReduce – Reduce Function (Reduce Step)**

[](https://cdn.journaldev.com/wp-content/uploads/2015/09/wordcount-mapping-reduce-step.png)

**MapReduce 3 Step Process With WordCount Example**

[](https://cdn.journaldev.com/wp-content/uploads/2015/09/mrv1-wordcount-complete-steps-e1441369662707.png)

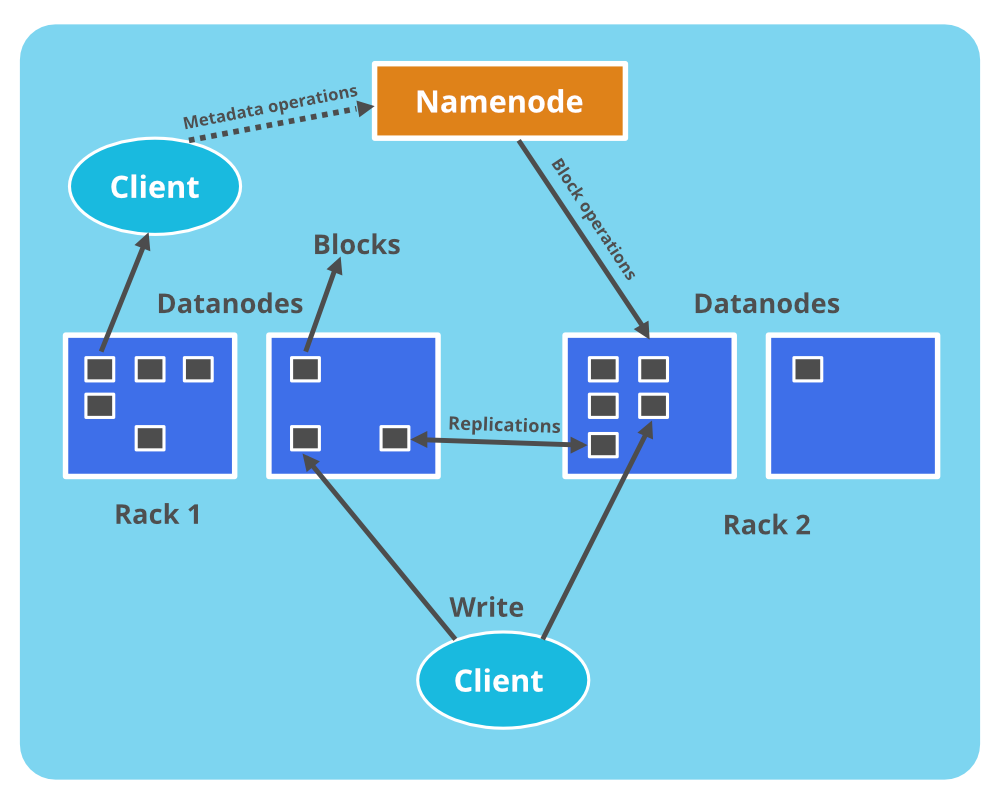
That’s it all about MapReduce Algorithm and map reduce example step by step.

In [Hadoop 1.x](http://data-flair.training/blogs/hadoop-introduction-tutorial-quick-guide/) NameNode was a single point of failure (SPOF). So, whole [Hadoop cluster](http://data-flair.training/blogs/install-configure-apache-hadoop-2-7-x-on-ubuntu/) becomes unavailable as soon as NameNode is down. In other words, [High Availability](http://data-flair.training/blogs/hadoop-high-availability-tutorial/) feature of the NameNode that talks about the necessity of a NameNode to be active for serving the requests of Hadoop clients is no more in existence in this scenario. As a result read operation fails.

[NameNode High Availability](http://data-flair.training/blogs/hadoop-hdfs-namenode-high-availability/) Architecture was introduced to solve this Single Point of Failure problem of NameNode. HA feature was intorduced in Hadoop 2.x where we have two NameNode in our HDFS cluster in an **active/passive** mode. The Active NameNode is the NameNode that works and runs in the Hadoop cluster. Passive NameNode similar to an active NameNode, it is also known asStandby NameNode. It comes into action only when the active NameNode fails. Whenever the active NameNode fails, the passive NameNode or the standby NameNode replaces the active NameNode, to ensure that the Hadoop cluster is never without a NameNode.The passive NameNode takes over the responsibility of the failed NameNode and keep the HDFS up and running. The passive Namenode takes the edit logs (meta data file) from NameNode and merges it with the **FsImage** (File system Image) to produce an updated FsImage as well as to prevent the Edit Logs from becoming too large. The read operation continues thereafter as if there was no downtime

# **How Does Namenode Handles Datanode Failure in Hadoop Distributed File System?**

Hadoop file system is a **master/slave** file system in which Namenode works as the master and Datanode work as a slave. Namenode is so critical term to Hadoop file system because it acts as a **central component** of HDFS. If Namenode gets down then the whole Hadoop cluster is inaccessible and considered dead. Datanode stores actual data and works as instructed by Namenode. A Hadoop file system can have multiple data nodes but only one active Namenode.



**Basic operations of Namenode:**

* Namenode maintains and manages the Data Nodes and assigns the task to them.
* Namenodde does not contain **actual data** of files.
* Namenode stores **metadata** of actual data like Filename, path, number of data blocks, block IDs, block location, number of replicas and other slave related informations.
* Namenode manages all the request(read, write) of client for actual data file.
* Namenode executes file system name space operations like opening/closing files, renaming files and directories.

**Basic Operations of Datanode:**

* Datanodes is responsible of storing actual data.
* Upon instruction from Namenode, it performs operations like creation/replication/deletion of data blocks.
* When one of Datanode gets down then it will not make any effect on Hadoop cluster due to **replication**.
* All Datanodes are synchronized in the Hadoop cluster in a way that they can communicate with each other for various operations.

#### What happens if one of the Datanodes gets failed in HDFS?

Namenode periodically receives a heartbeat and a Block report from each Datanode in the cluster. Every Datanode sends heartbeat message after every **3 seconds** to Namenode. The health report is just information about a particular Datanode that is working properly or not. In the other words we can say that particular Datanode is alive or not.  
A block report of a particular Datanode contains information about all the blocks on that resides on the corresponding Datanode. When Namenode doesn’t receive any heartbeat message for **10 minutes**(ByDefault) from a particular Datanode then corresponding Datanode is considered Dead or failed by Namenode. Since blocks will be under replicated, the system starts the replication process from one Datanode to another by taking all block information from the Block report of corresponding Datanode. The Data for replication transfers directly from one Datanode to another without data passing through Namenode.

## ****Standby NameNode: The Solution to NameNode Failure****

The HDFS high availability feature provides a facility of running two NameNodes in the same cluster. There is an active-passive architecture for NameNode; that is, if NameNode goes down, within a few seconds, the passive NameNode (also known as Standby NameNode) comes up. At any point in time, one of the NameNodes is in an Active state, and the other is in a Standby state. The Active NameNode is responsible for all client operations in the cluster, while the Standby NameNode is simply acting as a slave, maintaining enough state to provide fast failover if necessary.

For namespace information backup, the **fsImage** is stored along with the **editLog**. The editLog is like the journal ledger of NameNode. Through it, the in-memory fsImage can be reconstructed. It is needed to make the backup of editLog .

## ****What do you mean by the High Availability of a NameNode? How is it achieved?****

NameNode used to be single point of failure in Hadoop 1.x where the whole Hadoop cluster becomes unavailable as soon as NameNode is down. In other words, [***High Availability***](https://www.edureka.co/blog/how-to-set-up-hadoop-cluster-with-hdfs-high-availability/) of the NameNode talks about the very necessity of a NameNode to be active for serving the requests of Hadoop clients.

To solve this Single Point of Failure problem of NameNode, HA feature was intorduced in Hadoop 2.x where we have two NameNode in our HDFS cluster in an active/passive configuration. Hence, if the active NameNode  fails, the other passive NameNode can take over the responsibility of the failed NameNode and keep the HDFS up and running.

**Suppose there is file of size 514 MB stored in HDFS (Hadoop 2.x) using default block size configuration and default replication factor. Then, how many blocks will be created in total and what will be the size of each block?**

Default block size in Hadoop 2.x is 128 MB. So, a file of size 514 MB will be divided into 5 blocks ( 514 MB/128 MB) where the first four blocks will be of 128 MB and the last block will be of 2 MB only. Since, we are using the default replication factor i.e. 3, each block will be replicated thrice. Therefore, we will have 15 blocks in total where 12 blocks will be of size 128 MB each and 3 blocks of size 2 MB each.

**Name an apache system that implements a column database Hadoop**

**Ans:HBASE**